

Cloudproxy Nuts and Bolts

John Manferdelli¹

Overview

Cloudproxy is a software system that provides *authenticated* isolation, confidentiality and integrity of program code and data for programs even when these programs run on remote computers operated by powerful, and potentially malicious system administrators. Cloudproxy defends against observation or modification of program keys, program code and program data by persons (including system administrators), other programs or networking infrastructure. In the case of the cloud computing model, we would describe this as protection from co-tenants and data center insiders. To achieve this, Cloudproxy uses two components: a “Host System” (raw hardware, Virtual Machine Manager, Operating System) which provides capabilities described below to the protected program or “Hosted System” (VM, Application, Container).

A key concept for Cloudproxy is program measurement. A Host System measures a Hosted System incorporating the actual binary code of the Hosted System and configuration information affecting its execution resulting in a unique, globally descriptive, unforgeable, identity called the “Hosted System Measurement.” Since a Host System knows the “measurement” of each Hosted System it runs, it can store secrets that only that Hosted System will receive². The Host System can also “attest” to statements made by Hosted Systems by incorporating the Hosted System Measurement in statements it signs on behalf of the Hosted System. The upshot of this is that a Cloudproxy Hosted System can be isolated, can use secrets only it knows to encrypt and integrity protect all data it receives, sends or stores, as well as having secrets that allow it to securely authenticate itself over an otherwise unprotected network connection. Each Host System can itself be the Hosted System of a parent Host System and each such “child Host System” relies on its parent Host System to protect the keys the child Host System uses to provide services for its Hosted Systems. The “root” or “base” Host System is hardware which, fortunately, is widely available.

Learning how to develop Cloudproxy applications is easily achieved by seeing and understanding fairly simple working code. We developed a simple application, called *simpleexample*, that is used throughout this paper. Reading this paper in conjunction with understanding *simpleexample* should enable you to develop Cloudproxy applications within a day or so. The source code for *simpleexample* is in the Cloudproxy distribution.

¹ John is manferdelli@google.com or jlmucbmath@gmail.com. Cloudproxy is based on work with Tom Roeder (tmroeder@google.com) and Kevin Walsh(kevin.walsh@holycross.edu) and Fred Schneider.

² To do this, the Host System must be isolated and have access to secrets only it knows. Given that, the Host System, using its secrets, simply encrypts and integrity protects the Hosted System secrets along with the Hosted System Measurement. It only decrypts those secrets for a Hosted System with the same Hosted System Measurement.

Readers can consult [1] for a fuller description. Source code for Cloudproxy as well as all the samples and documentation referenced here is in [2].

Principal Names in Cloudproxy

Principal names in Cloudproxy (called “Tao Principal Names”) are globally unique, general and can represent key based principals, machine based principals, and, most importantly, program based principals. A Tao Principal Name is hierarchical and, in the case of program principals, contains the Hosted System Measurement as well as the measurement of all its Host and all its Host System’s ancestors. For example, a principal rooted in a public key will have the cryptographic hash of it in its name making it globally unique and unforgeable. Since a program principal has the Hosted System Measurement in its Tao Principal Name as well as the measurement of the Host System and all its ancestors, it too is globally unique and unforgeable

The name for a Hosted System (i.e., a program) running on a “base” Host System identified by a base public key (typically a hardware key) might look something like

```
key([080110011801224508011241046cdc82f70552eb...]).Program([25fac93bd4c  
c868352c78f4d34df6d2747a17f85...])
```

Here, `key([080110011801224508011...])` represents the signing key of the (hardware) Host System³ and `Program([25fac93bd4cc868352c78f4d34df6d2747a17f85...])` is a cryptographic hash of the code and configuration information of the booted Hosted System⁴. If the Host System were a Linux host rooted in a Trusted Platform Module (“TPM”) authenticated boot, its name would include the hash of the TPM’s AIK at the top level of hierarchy, the PCRs of the booted Linux system, the hash of the Authenticated Code Module (“ACM”)⁵ that initiated the authenticated boot and the hash of the Linux image and its initramfs⁶. If the Hosted System was a Linux application whose Host System was a Cloudproxy enabled Linux which was hosted by a Hypervisor that booted under a TPM boot, its name would include the AIK at the top level of the hierarchy, the PCR values and boot flags of the hypervisor at the second layer of the hierarchy, the measurement (performed by the hypervisor) of the Linux OS (again naming its code hash, boot flags and initramfs hash), as well as the measurement (made by the Linux host) of the application code and configuration information (like command arguments) at the third level of the name hierarchy.

In the *SimpleExample* execution output, analyzed below, there are many more examples of Tao Principal Names.

³ The byte value in key is a SHA-256 cryptographic hash of the corresponding public key serialized in an internal key format. So, if you know the public key, you can verify that the bytes in the key name correspond to that self-same public key.

⁴ As indicated by the ellipsis (“...”) principal names are often longer and may even contain.

⁵ The ACM is a piece of software that controls the authenticated boot of a TPM mediated boot, so it must be included as “configuration information.”

⁶ Initramfs will have security critical code like the service that implements the Tao so it must be measured along with the kernel image to provide an accurate identity for the “running Linux OS.”

The Cloudproxy API

The Cloudproxy programming model is simple and uses only a few API calls. Cloudproxy provides a programming interface in Go or C++ and we refer to the collection of Cloudproxy API's as the "Tao Library."

There are two Cloudproxy principal API interfaces of interest for Go programmers. The first is the Tao API to access functions in a Hosted System. The interface definition is in `$CLOUDPROXYDIR/go/tao/tao.go`. The interface functions are:

`GetTaoName()` returns the Tao principal name of the Hosted System.

`ExtendTaoName(subprin auth.SubPrin)` irreversibly extends the Tao Principal Name by adding an additional node to the hierarchical Tao Principal Name.

`GetRandomBytes(n int)` returns `n` cryptographically secure random bytes.

`Rand()` returns an `io.Reader` for random bytes from this Tao.

`Attest(issuer *auth.Prin, time, expiration *int64, message auth.Form)` returns an Attestation from the Host System. The (optional) issuer, time and expiration will be given default values if nil; the message attested to is a form in the Tao authorization language.

`Seal(data []byte, policy string)` returns a blob encrypted and integrity protected by the Host System containing the data, policy and measurement of the Hosted System in a form suitable for `Unseal`.

`Unseal(sealed []byte)` decrypts and returns the data and policy string, if access complies with the policy which usually includes having the embedded Hosted System measurement match the embedded measurement.

There are a few additional Tao functions, commonly used by Hosted Systems, related to domain management, and guards employed for authorization decisions:

`CreateDomain(cfg DomainConfig, configPath string, password []byte) (*Domain, error)`

`DomainLoad` is used to store and retrieve Program Certificates and sealed policy data.

`Parent()` which gets the parent interface to the Tao. If `t := tao.Parent()`, for example, we'd call `Attest` as `t.Attest(issuer, time, expiration, message)`.

`DialTLS(network, addr string)` creates a new X.509 certs from fresh keys and dials a given TLS, returns connection.

`DialWithKeys(network, addr string, guard tao.Guard, v *tao.Verifier, keys *tao.Keys)` connects to a TLS server using an existing set of keys, returns `net.Conn`.

`Listen(network, laddr string, config *tls.Config, g tao.Guard, v *tao.Verifier, del *tao.Attestation)` returns a new Tao-based `net.Listener` that uses the underlying crypto/tls `net.Listener` and a `Guard` to check whether or not connections are authorized.

`ValidatePeerAttestation(a *Attestation, cert *x509.Certificate, guard Guard)` checks a `Attestation` for a given `Listener` against an X.509 certificate from a TLS channel.

`(l *anonymousListener) Accept() (net.Conn, error)` `Accept` waits for a connect, accepts it using the underlying `Conn` and checks the attestations and the statement.

The Tao Library also contains helper functions to build and verify Program Certificates, perform common crypto tasks like key generation and establish the Tao Channel. These are most easily understood by looking at the *simpleexample* code below.

Finally, the Tao Library has rather extensive and flexible authorization support. Authorization decision are performed by *guards*.

Current *guards* include:

- The liberal guard: this guard returns true for every authorization query
- The conservative guard: this guard returns false for every authorization query
- The ACL guard: this guard provides a list of statements that must return true when the guard is queried for these statements.
- The datalog guard (used in the example below): this guard translates statements in the CloudProxy auth language (see `$CLOUDPROXYDIR/go/tao/auth/doc.go` for details) to datalog statements and uses the Go datalog engine from `github.com/kevinawalsh/datalog` to answer authorization queries. See `install.sh` for an example policy.

A brief description of the guards and authorization language appears in appendix 1 but you don't need to understand the authorization language to understand *simpleexample*.

Examples of all these calls (and their arguments), except for `Rand`, appear in the *simpleexample* code.

The second API is the Host API used by Host Systems and defined in `$CLOUDPROXYDIR/go/tao/host.go`. It consists of the following calls:

`GetRandomBytes(childSubprin auth.SubPrin, n int)` which returns random bytes

Attest(childSubprin auth.SubPrin, issuer *auth.Prin, time, expiration *int64, message auth.Form) which requests the Host System's Host sign a statement on behalf of

Encrypt(data []byte) returns and encrypted, integrity protected blob only the Host can decrypt.

Decrypt(encrypted []byte) returns the data protected by Encrypt if this is the right Host.

AddedHostedProgram(childSubprin auth.SubPrin) notifies host that a new Hosted System has been created.

RemovedHostedProgram(childSubprin auth.SubPrin) notifies the Host that a Hosted System has been terminated.

HostName() returns the Principal Name of this Host System.

The best way to learn Cloudproxy is by examining the annotated code in `$CLOUDPROXYDIR/go/apps/simpleexample` in conjunction with the commentary below.

The Tao Paradigm

The Tao is often used in a stereotypical way, which we refer to as the Tao Paradigm. Cloudproxy programs always have policy public keys embedded (PK_{policy}) in their image either explicitly or implicitly⁷. Statements signed by the corresponding private key (pK_{policy}), and only those statements, are accepted as authoritative and acted on by these programs. The policy key(s) plus the Hosted System code and configuration, reflected in its measurement, fully describe how the Hosted System should behave and, hence, an authenticated measurement is a reliable description of expected behavior.

In the Tao Paradigm, when a program first starts on a Hosted System, it makes up a public/private key-pair ($PK_{program}/pK_{program}$) and several symmetric keys that it uses to "seal" information for itself. A Hosted System then "seals," using the Host System interface, all this private (key) information⁸. After that, the Hosted System requests an Attestation from its Host System, naming the newly generated $PK_{program}$ and sends the resulting Attestation to a security domain service which confirms the security properties in the Attestation and Host Certificate⁹. If the Attestation and Host Certificate meet security domain requirements, the security domain service signs (with pK_{policy}) an x509 certificate specifying $PK_{program}$ and the Tao Principal Name of the Hosted System. The resulting certificate, called the *Program Certificate*, can be used by any Hosted System to prove its identity to another Hosted System in the same security domain.

⁷ Explicitly embedding the key just means that it appears in initialized data measured as part of the program. An example of implicit embedding, which is described in more detail below, is reading in, say, the policy key and extending the identity of the program with that key.

⁸ The sample code in `$CLOUDPROXYDIR/go/apps/fileproxy` demonstrates rollback protection for this sealed data and we plan to improve local rollback support in the near future.

⁹ The actual attestation being signed by the Host System expressed in a formalized language is $PK_{program}$ speaks for the Hosted System's Tao Principal Name.

Program Certificates are used to negotiate encrypted, integrity protected TLS-like channels between Hosted Systems (the “Tao Channel”). A Hosted System can share information over these channels with full assurance of the code identity and security properties of its channel peer. Once established, each endpoint of the Tao Channel “speaks for” its respective Hosted System.

Hosted Systems in the same security domain can fully trust other authenticated Hosted Systems in that security domain with data or processing. Typically, a Hosted System uses the symmetric keys it generates and seals at initialization to encrypt and integrity protect information it stores on disks or remotely.

The Tao Paradigm also often uses a “domain service” to maintain the policies of a distributed Cloudproxy application. Employing a centralized security domain service eliminates the need for each and every Cloudproxy Hosted System in a security domain to maintain lists of trusted hardware or trusted programs and simplifies distribution, maintenance and upgrade.

Often, Hosted Systems in the same security domain will share intermediate keys to protect data that may be used under many Host System environments. These keys can be shared based on policy-key signed directives as Host or Hosted Systems are upgraded or new systems are introduced in a controlled but flexible way eliminating the danger that data might become inaccessible if a particular Cloudproxy system is replaced or becomes damaged or unavailable. The penultimate section of this paper discusses key management techniques as keys, programs and domain information changes over the lifetime of a Cloudproxy based application.

Hardware roots of Trust

Cloudproxy requires that the lowest level, “booted” system software (the “base system”) be measured by a hardware component which provides attest services, seal/unseal services and, usually, some hardware facilities to isolate Hosted Systems. Absent hardware protection, remote users have no principled way to trust the security promises (isolation, confidentiality, integrity, verified code identity) of Cloudproxy since “insiders” might have silently changed security critical software or stolen low level keys.

Cloudproxy supports TPM 1.2 and TPM 2.0 as hardware roots for Host Systems. We have implemented support for other mechanisms and believe adding a new hardware mechanism is relatively easy. In addition, Cloudproxy also can initialize and run on a “soft Tao” which simulates secure base system protection. This allows for easy debugging on a wider variety of platforms.

Once the base Host System is safely measured and booted on supported hardware, Cloudproxy provides for recursive Host Systems at almost every layer of software including:

1. A Host System consisting of hardware (e.g. - TPM, SMX) that hosts a VMM which isolates Hosted Systems consisting of Virtual Machines.

2. A Host System running in an operating system which isolates Hosted Systems consisting of processes (or applications).
3. A Host System running in an operating system which isolates Hosted systems hosted consisting of subordinate Operating Systems or Containers.
4. A Host System running in an application (like a browser) which isolates Hosted Systems consisting of sub-applications, like plug-ins.

In all cases, Hosted Systems employ the same Cloudproxy API and can use any non-Cloudproxy native service (for example, any system call in Linux) so the programming model at each Hosted System layer is essentially unchanged from corresponding non-Cloudproxy code.

Sample Applications

This paper is intended to allow you to use Cloudproxy immediately on a Linux based Cloudproxy Host System. To this end we include installation instructions for running under a “soft Tao.” Full instructions for installing and deploying Cloudproxy based systems on TPM 1.2 and TPM 2.0 Intel hardware with SMX extensions appears in a companion paper called “Cloudproxy Nuts and Bolts Deploying Real Cloudproxy Applications”[6].

You do not need to read the deployment guide to learn to write Cloudproxy applications. This paper is self-contained; in it, we use a standard simple application called, cleverly, *simpleexample* throughout this paper.

Installing Cloudproxy on Linux or a Mac

First, you should download the Cloudproxy repository from [2]. To do this, assuming you have git repository support, type

```
git clone https://github.com/jlmucb/cloudproxy,
```

or,

```
go get https://github.com/jlmucb/cloudproxy.
```

This latter command will also install the needed go libraries. You can also download a zipped repository from github. You should probably install this in `~/src/github.com/jlmucb` (which we refer to as `$CLOUDPROXYDIR`) to save go compilation problems later. It's a good idea to put go binaries in `~/bin` as is common in Go. Follow the installation instructions in `$CLOUDPROXYDIR/Doc`; that directory also contains [1] and an up to date version of this document as well as installation instructions for TPM 2.0 capable machines and installation instructions for Cloudproxy enabled KVM hypervisors and Docker containers.

You must also install the Go development tools (and C++ development tools if you use the C++ version) as well as *protobuf*, *gtest* and *gflags* as described in the Go documentation.

Next, compile, and initialize the *SimpleExample* application in

`$CLOUDPROXYDIR/go/apps/SimpleExample` and run it as described in the next section.

Understanding Simple Example

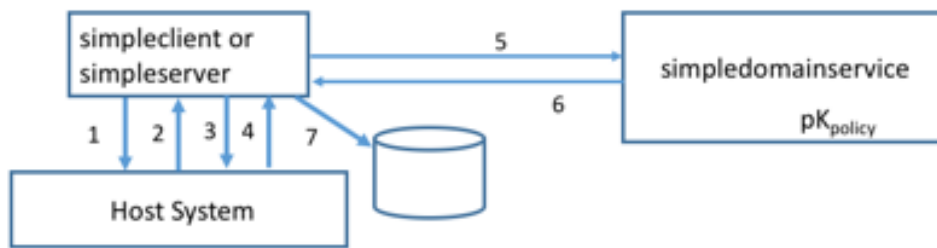
There are three application components in *simpleexample*, each producing a separate executable:

1. A Simple Client in `$CLOUDPROXYDIR/go/apps/SimpleExample/SimpleClient/simpleclient.go`.
2. A Simple Server in `$CLOUDPROXYDIR/go/apps/SimpleExample/SimpleServer/simpleserver.go`.
3. A Simple Security Domain Signing Service in `$CLOUDPROXYDIR/go/apps/SimpleExample/SimpleDomainService/simplesdomainsevice.go`.

Common code used by the client and server is in

`$CLOUDPROXYDIR/go/apps/SimpleExample/taosupport`.

When *simpleclient* and *simpleserver* start for the first time on the Host System, they generate a public/private key pair called a program key. Program Keys authenticate the program. In order to be useful, program keys must be “trusted” by other domain components. This is achieved, in the Tao Paradigm by providing attestations of the program public keys to *simplesdomainservice*. If the measurements are correct, *simplesdomainservice* signs Program Certificates for those keys with pK_{policy} . This interaction is depicted below.



Initialization

1. simpleclient (or simpleserver) generates public/private key pair $PK_{\text{simpleclient}}$, $pK_{\text{simpleclient}}$. simpleclient requests Host System attest $PK_{\text{simpleclient}}$.
2. Host System returns attestation.
3. simpleclient generates additional symmetric keys and request Host System seal symmetric keys and $pK_{\text{simpleclient}}$.
4. Host System returns sealed blobs.
5. simpleclient connects to simpliedomainservice and transmits attestation.
6. simpliedomainservice returns signed Program Certificate.
7. simpleclient stores sealed blobs and Program Certificate for later activations.

Initialization

After initialization of keys and secrets, *simpleserver* waits for *simpleclient* instances to request their secret (which is client dependent). Each *simpleclient* instance uses a Tao Channel to contact the *simpleserver* to learn its secret. We don't implement rollback protection or distributed key management for intermediate secrets in *simpleexample* just to keep the example as simple as possible. *simpliedomainservice* is the domain service for *simpleexample*.



Operation

1. Simpleserver reads previous sealed blobs and Program Certificate.
2. Simpleserver requests Host System unseal blobs yielding symmetric keys and private program key.
3. Host system returns unsealed blobs.
4. Simpleclient reads previous sealed blobs and Program Certificate.
5. Simpleclient requests Host System unseal blobs yielding symmetric keys and private program key.
6. Host system returns unsealed blobs.
7. Simpleclient and simpleserver open encrypted, integrity protected channel using their program keys and certificates.
8. Simpleclient transmits a request to retrieve secret.
9. Simpleserver retruns secret.

Operation

We describe the Cloudproxy API, compilation and installation, execution and output of the Go version of *simpleexample* in the sections below. We also provide a general description of the critical Cloudproxy elements used in the *simpleexample* to help you get used to the programming model. Since the domain service does not use the Cloudproxy API extensively, we don't describe code in detail here although

`$CLOUDPROXYDIR/go/apps/SimpleExample/simplesdomainservice` contains a full working version. A corresponding C++ version of *simpleclient* is described in an appendix.

Although *simpleexample* is in fact very simple, the Tao relevant code in *simpleexample* can be used with little change, even in complex Cloudproxy applications.

Simple Client in Go

simpleclient is implemented as a single go file in

`$CLOUDPROXYDIR/go/apps/simpleexample/simpleclient/simpleclient.go`. Another file, `$CLOUDPROXYDIR/go/apps/simpleexample/taosupport`, provides some common support code used by *simpleclient* and *simplserver*. You should open those files and read the code as you read this.

When it starts, *simpleclient* parses the flags it was called with and calls `TaoParadigm` (a common support function which establishes the “Tao Paradigm” in `$CLOUDPROXY/go/apps/simpleexample/taosupport/taosupport.go`). As with *simpleclient*, *simpleserver* calls `TaoParadigm` with the location of its domain configuration information, the directory in which *simpleclient* can save its local files (like sealed keys and certs) and a preallocated `TaoProgramData` object. If successful, `TaoParadigm` returns a filled `taosupport.TaoProgramData` object containing the policy cert for the domain and *simpleclient*’s Tao Principal Name, symmetric keys, Program Key and Program Cert. Almost all the Cloudproxy specific code is concentrated in `TaoParadigm` which uses the Tao API; it’s described below. Note that *simpleclient* erases its keys (`defer taosupport.ClearTaoProgramData(&clientProgramData)`) after completing the request.

After calling `TaoParadigm`, *simpleclient* calls `OpenTaoChannel` (another support function in `taosupport.go`) to open up a Tao Channel with *simpleserver*. If successful, `OpenTaoChannel` returns the channel stream as well as the Tao Principal Name of the *simpleserver* it connected to.

Finally, *simpleclient* sends a request for its secret and calls `GetResponse` to retrieve *simpleserver*’s response. If the request was successful, it prints its secret.

Simple Server in Go

simpleserver is implemented as a single go file in `$CLOUDPROXY/go/apps/simpleexample/simpleclient/simpleserver.go` and also employs the support functions in `$CLOUDPROXY/go/apps/simpleexample/taosupport`.

Just as *simpleclient*, *simpleserver* parses the flags it was called with and calls `TaoParadigm` with the location of its domain configuration information, the directory in which *simpleserver* can save its local files and a preallocated `TaoProgramData` object. As above, `TaoParadigm` returns a filled `taosupport.TaoProgramData` containing the policy cert for the domain and *simpleserver*’s Tao Principal Name, symmetric keys, Program Key and Program Cert.

simpleserver sets up cert chain rooted in the Policy Certificate to validate Program Certificates it received from peer clients, and listens over the indicated socket. In addition to the client Program Certificate, the TLS handshake uses *simpleserver*’s Program Certificate. After initialization, *simpleserver* then loops through a standard socket acceptance loop, waiting for successful client connections. Each successful connection is serviced in a separate thread. A successful client connection will result in a valid peer Certificates for the corresponding *simpleclient* obtained and verified by the TLS handshake. *Simpleserver* retrieves extracts the *simpleclient*’s Tao Principal Name from the *simpleclient* Program Certificate (which appears in

the Organizational Unit of its x509 name) and dispatches a thread [`go serviceThead(ms, clientName, serverProgramData)`] with the per-client service channel, client Principal Name and *simpleserver's* `TaoProgramData` object.

`serviceThead` loops through a standard service request-response loop. It calls `taosupport.GetRequest` on the designated service channel to get requests. If it receives a valid request, it calls `HandleServiceRequest` to service the request. There is only one valid request which is “give me my secret,” after receiving such a request, `HandleServiceRequest` makes up a client specific secret consisting of the the `clientName` with “43” appended and return is using `taosupport.SendResponse`. After the first successful request, *simpleserver* terminates.

Common support code

Now we describe the common code used by *simpleserver* and *simpleclient* employing the Cloudproxy Tao.

`TaoParadigm` loads the domain information from the provided configuration file [`err := tao.LoadDomain(*cfg, nil)`] and retrieves the policy certificate from the domain information. It hashes the policy cert and uses this to Extend the Tao Principal Name which binds the policy key to the current program identity.

Next `TaoParadigm` calls `LoadProgramKeys`, which retrieves previously created sealed symmetric keys and Program Key along with the Program Certificate (if they exist), from the application directory, otherwise it returns nil. If the sealed sealed symmetric keys were recovered, it unseals them, otherwise (via the call to `InitializeSealedSymmetricKeys`), it generates new keys, seals them and saves them to the correct file in the application directory. If the sealed Program Key exists, it unseals the Program Key, otherwise, `TaoParadigm` generates a new program key, builds a request for *simplesdomainserver*, attests the new Program public key and transmits the request and attestation to *simplesdomainserver* to have a Program Certificate signed by the policy key. This is done by `InitializeSealedProgramKey` which we describe further below. Before returning, `InitializeSealedProgramKey`, seals the private Program Key, and stores the sealed key and certificate in the application area and returns the new Program Key and Program Certificate.

Finally, `TaoParadigm` fills the `TaoProgramData` object with the symmetric keys, program key, policy certificate, program certificate and the location of application store; it then returns.

`InitializeSealedProgramKey` carries out the heart of the Cloudproxy key management service, so it and its callees are worth a little further discussion.

`InitializeSealedProgramKey` calls

`CreateSigningKey` which generates a new Program Key, builds an x509 certificate request which includes the Hosted System's Tao Principal Name, and self-signs that certificate request. It then constructs a statement in the authorization language that says "PrincipalName(Program-Key) speaksfor PrincipalName(Program)" where each "PrincipalName" is the canonical Tao Principal name of, respectively, the new key and the Program's Tao Principal Name. This is called, a delegation statement. `CreateSigningKey` then requests the Host System (via the Tao Interface) attest to the delegation statement. The attestation includes the delegation, the Measurement of the Hosted System and is signed by the Host System. The resulting attestation means "HostSystem(hash-of-attestation-key) says PrincipalName(Program-Key) speaksfor PrincipalName(Program)." This attestation along with any relevant supporting certificates, is transmitted to the domain signing service (*simpledomainservice*) via the call to `RequestDomainServiceCert`. The domain service, if the Hosted System measurement conforms to the list of "trusted programs" in the domain, signs the Program Certificate with the (private portion of) the policy key and returns it. The resulting Program Certificate means "Policy-Key says PrincipalName(Program-Key) speaksfor PrincipalName(Program)." Any program in the domain receiving the Program Certificate from a communicating program can verify the Program Certificate (using the public portion of the policy) and demand the communicating program "prove possession" of the private portion of the Program Key. Such a proof cryptographically authenticates the communicating program and the Tao properties under which it was created.

Two programs in the security domain, one acting as a client and one acting as a server, use their Program Keys to open an encrypted, mutually authenticated, integrity protected TLS channel. Once this channel is established each program know "the channel speaks for the peer Program Principal." The client side of this channel negotiation is accomplished by the program `OpenTaoChannel`. It simply uses the Program Key of the client (and the received Program Certificate of the server which is authenticated by the policy key) to open the TLS channel. `OpenTaoChannel` returns the resulting bidirectional channel handle and the Tao Principal Name of the server. The corresponding code in the server to open the channel is in *simpleserver* and uses its Program Key and Program Certificate.

`GetRequest`, `SendRequest`, `GetResponse`, `SendResponse` are simple helper functions to get and send requests and responses. `Protect` and `Unprotect` are simple functions to encrypt and decrypt files protected by a Hosted System's symmetric keys.

A Hosted System can also ask the Host System to measure and start other programs. We used a utility (*tao run*, see below) to start *simpleclient*, *simpleserver* and *simpledomainservice* during initialization (and described below) so there was no need to do this in the sampleexample code. Starting a Hosted System varies a little depending on the Host System environment. To see how this is done in programmatically Linux, consult `$CLOUDPROXYDIR/go/apps/tao_launch`.

We should mention that *simpleexample* was meant mainly to be instructive (but correct!) so we sometimes repeated code that could have been accessed in the Tao Library. We also opted for simple, transparent constructions in Go sometimes at the expense of being “idiomatically correct.” The `TaoProgramData`, for example, duplicates some data structures in the Tao Library and is defined to simplify and clarify the actual Tao Data but it can be replaced.

We don’t describe *simpledomainservice* here since it does not directly call the Tao interface. You can find other example applications in `$CLOUDPROXYDIR/go/apps/demo` and a more complex example in `$CLOUDPROXYDIR/go/apps/fileproxy`.

Finally, an appendix has a brief description of the Datalog policy engine and rules.

Configuring, compiling and running SimpleExample

When the Tao Host System starts, it requires several kinds of information:

- A public key that roots the *Tao* on the hardware,
- Host data, including the mechanism used to communicate between the Hosted System and the Host System, rules affecting which Hosted Systems the Host System should run, and, in the case of a hardware rooted Host System, the hardware mechanism that is employed (e.g., TPM 1.2 or TPM 2.0).
- Domain data (in our case for the *simpleexample* domain) including the policy key and corresponding private key, and the self-signed policy cert.

In addition, we need an implementation for the “Host System” which includes support for the host including support the host’s isolation mechanism (processes in Linux) and communications channels used to communicate with Hosted Systems. In our case, the Host System is Linux and the implementation (whether using a soft tao, TPM 1.2 or TPM 2.0) is *linux_host*¹⁰.

The public key rooting the hardware tao is usually produced by a TPM utility; in the TPM 1.2 nomenclature, this is called the AIK. The public key rooting the TPM 2.0 is the endorsement key. In our demo, we use a “soft tao” which is rooted in a key.

The Host Data consists of the Host Certificate (used to validate nested Host System Attestation) and it’s sealed keys and data, stored in

```
$CLOUDPROXYDIR/go/apps/SimpleExample/SimpleDomain/domain.simpleexample/linux_
tao_host.
```

The Domain data includes the policy key and corresponding (encrypted) private key, hostname, host type and communications channel used between the Host System and its Hosted Systems, information related to the guards used¹¹ as well as signatures over the binaries that are part of

¹⁰ *linux_host* is also the implementation used by a KVM Host System and Docker containers and in fact, all Host Systems running on Linux.

¹¹ Look at `$CLOUDPROXYDIR/go/tao/domain.go` for further details.

the domain (if the Host System limits what Hosted Systems it will run, in our case, these are the *simpleclient* and *simpleserver* binaries).

In *simpleexample*, all domain information is contained in files in `/Domains/domain.simpleexample/linux-tao/*`. Other sub-directories of `/Domains/domain.simpleexample`, namely, *SimpleClient*, *SimpleServer* and *SimpleDomainService* contains data files stored and retrieved by these programs like sealed keys and Program Certificates.

There is a single utility, called *tao*, which initializes this domain data, activates the tao host and runs the applications. We provide shell scripts to call *tao* with the right arguments, these scripts are in `$CLOUDPROXYDIR/go/apps/simpleexample/SimpleDomain`. These scripts use several path variables, namely:

```
TAO_HOST_DOMAIN_DIR=~/.src/github.com/jlmuch/cloudproxy/go/apps/simpleexample/SimpleDomain
OLD_TEMPLATE=$TAO_HOST_DOMAIN_DIR/domain_template.simpleexample
DOMAIN=/Domains/domain.simpleexample
TEMPLATE=/Domains/domain_template.simpleexample
BINPATH=~/.bin
```

Domain templates configure domain information. We have provided a sample template in *SimpleDomain/domain_template.simpleexample*. However, you can generate such a template by running *gentemplate*. “*gentemplate*” consists of:

```
$BINPATH/tao domain init -tao_domain $DOMAIN -config_template $TEMPLATE -
pass "xxx"
/home/jlm/src/github.com/jlmuch/cloudproxy/go/run/scripts/domain_template
.pb > $TEMPLATE \
sed "s/REPLACE_WITH_DOMAIN_GUARD_TYPE/Datalog/g"
```

This template contains information included in the policy cert, the rules used by the domain when authenticating images and the location of the images which must be measured and recorded in the policy rules.

To initialize *simpleexample*, first, we must initialize the directory that will hold domain information. Because of a limitation in one of the Go libraries, Domain paths cannot be too large. We keep all our domains in `/Domains` and we keep the *simpleexample* domain information in `/Domains/domain.simpleexample`, although you can put them elsewhere as long as the path name is short enough.

To do this, we do

```
mkdir /Domains
```

(if that directory does not already exist) and then call *initidomainstorage*. “*initidomainstorage*” sets up the storage hierarchy; it consists of:

```
#
source ./defines
```

```

if [ -e $DOMAIN ]
then
    ls -l $DOMAIN
else
    mkdir $DOMAIN
fi
cp $OLD_TEMPLATE $TEMPLATE
source ./defines
if [[ -e $DOMAIN/SimpleClient]]
then
    echo "$DOMAIN/SimpleClient exists"
else
    mkdir $DOMAIN/SimpleClient
    echo "$DOMAIN/SimpleClient created"
fi
if [[ -e $DOMAIN/SimpleServer]]
then
    echo "$DOMAIN/SimpleServer exists"
else
    mkdir $DOMAIN/SimpleServer
    echo "$DOMAIN/SimpleServer created"
fi
if [[ -e $DOMAIN/SimpleDomainService]]
then
    echo "$DOMAIN/SimpleDomainService exists"
else
    mkdir $DOMAIN/SimpleDomainService
    echo "$DOMAIN/SimpleDomainService created"
fi

```

To initialize the (soft) host key, call *initkey*, which does the following:

```

#
source ./defines
if [[ -e $DOMAIN/linux_tao_host ]]
then
    echo "$DOMAIN/linux_tao_host exists"
else
    mkdir $DOMAIN/linux_tao_host
    echo "$DOMAIN/linux_tao_host created"
fi
KEY_NAME="$($BINPATH/tao domain newsoft -soft_pass xxx -config_template
$TEMPLATE $DOMAIN/linux_tao_host)"
echo "host_name: \"$KEY_NAME\"" >> $TEMPLATE

```

“newsoft” generates a new soft key. The arguments following the flags “-config_template -tao_ -pass” specify, respectively, the location of the template, the location where the domain information is stored and the password protecting the private policy key. This produces the xxx file containing root Tao key. To use a TPM based Host System, you’d call a corresponding program to put the AIK in template.

After we initialize the domain and host keys, we compile *simplifiedomainservice*, *simpleserver*, *simpleclient*. First, we compile the programs comprising simpleexample, namely *simplifiedomainservice*, *simpleserver*, and *simpleclient*. To do this we type

```
go build ...
```



```
go install ...
```

These build the images and puts them in `~/bin`. Copy these files into `/Domains`.

To run `simpleexample`, we provide a number of scripts. All the scripts should be run as root. We describe each step but we have collected them in single *runall* script. So you can just type *runall* to run (as root).

We need to initialize the domain information. To initialize the domain, call *initdomain* which consists of the following:

```
#
source ./defines
$BINPATH/tao domain init -tao_domain $DOMAIN -config_template $TEMPLATE -
pub_domain_address "127.0.0.1" -pass xxx
$BINPATH/tao domain policy -add_host -add_programs -add_linux_host -
add_guard -tao_domain \
    $DOMAIN -pass xxx -config_template $TEMPLATE
```

The first call produces the files in `$DOMAIN/linux_tao_host/{cert,keys,host.config}`. The second measures the applications in the domain if the template specifies a more restrictive program run policy than “allow anything to run”; these programs should have previously been copied into `/Domains`.

To initialize the (Linux) host, call *inithost* which does the following:

```
$BINPATH/tao host init -tao_domain $DOMAIN -hosting process -root -pass
xxx
```

This generates linux host configuration information stored in the `$DOMAIN/SimpleDomain/domain.simpleexample/linux_tao_host` directory. The argument to the “-hosting” flag is the kind of child hosts, namely, Linux processes. The “-root” flag means this is a “root” host (i.e. – the lowest level Tao). For hosts stacked on other hosts, we would use the “-stacked” flag. For example,

```
$BINPATH/tao host init -tao_domain $DOMAIN -hosting process -stacked -
parent_type tpm
```

To run our development Host System¹², call *runhost*, which consists of:

```
$BINPATH/tao host start -tao_domain $DOMAIN -host linux_tao_host/ -pass
xxx &
```

The argument to the “-host” flag is the subdirectory of `SimpleDomain/domain.simpleexample` that contains the host information.

Finally, to run a Hosted System, like *simpleclient*, we would say:

¹² In a production system, the Host System would have already been started in the host’s initialization scripts.

```
$BINPATH/tao run $BINPATH/simpleclient -tao_domain $DOMAIN &
```

Some further observations: The password supplied in the calls to `tao domain init` and `tao domain policy protect` access to the policy private key. The password supplied to the `tao newsoft`, `tao host init` and `tao host run` protect the soft host private key; this password is usually not the same as the key protecting the policy private key. `linux_host`, which implements the Linux Host, uses the rules generated by `tao domain policy` to decide whether to run an application.

Often, a Host System does not use application security domain rules to determine what to run and this is how our example is constructed.

where `x` is the measurement of the `linux_host` or an “AllowAll” policy. As you become familiar with the Datalog rules, you can apply them flexibly but that is a distraction in SimpleExample.

To summarize, to run the tests:

```
cd ~/src/github.com/jlmucb/cloudproxy/go/apps/simpleexample/SimpleDomain
./gentemplate
./initdomainstorage
./initkey
go build ...
go install ...
cp ~/bin/{SimpleSomainService, SimpleServer, SimpleClient} /Domains
sudo bash
./runall
```

After running the programs, copy the *clean* script from

`~/src/github.com/jlmucb/cloudproxy/go/apps/simpleexample/SimpleDomain` into `/Domains/domain_simpleexample`, `cd` to the `/Domains/domain_simpleexample` and run *clean* from there. Make sure you kill previous instances of *linux_host*, *simpledomainservice*, *simpleserver* and *simpleclient* before re-running *runall*; you need not rerun *gentemplate*, *initdomainstorage* or *initkey*.

What the output from SimpleExample teaches us about the Tao

The most concrete way to understand Cloudproxy is to follow the code example and the output. Here is a brief description of the output of the Go version of SimpleExample using a “soft” tao.

In our execution setup, recall that the domain information is in `/Domains/domain.simpleexample`; this includes the template, tao prepared configuration files and three directories: `SimpleClient`, `SimpleServer` and `SimpleDomainService` which are directories in which application information (mostly sealed keys) are stored for,

respectively, `SimpleClient`, `SimpleServer` and `SimpleDomainService`. Binaries are stored in the directory `~/bin` as is customary in go.

In the repository, there are also three shell scripts to facilitate running the examples. The script `compile` compiles the applications and puts them into `bin`. After making the directory, `/Domains`, use `initdomainstorage` to initialize the storage areas. Copy the script `clean` into `/Domains/domain.simpleexample/SimpleDomain` and make it executable. If you employ a restrictive program running program don't forget to recompile the programs and copy them into `/Domains` before running `initdomain`. **Then, re-run `runall`** to run `SimpleExample`. After it runs, you can run `clean`, in `/Domains/domain.simpleexample/SimpleDomain`, to erase the output files. `clean` runs a `ps aux | fgrep simple` at the end to tell you what lingering processes to kill (`kill -9`) so you can run subsequent tests.

Our example uses the Datalog authorization subsystem so system rules are expressed in the Datalog policy language. Example statements in Datalog can be seen in the template file.

When you run `simpleexample`, look at the output; you'll notice, at the beginning:

```
Warning: Passwords on the command line are not secure. Use -pass option
only for testing.
Warning: Passwords on the command line are not secure. Use -pass option
only for testing.
Warning: Passwords on the command line are not secure. Use -pass option
only for testing.
Warning: Passwords on the command line are not secure. Use -pass option
only for testing.

Linux Tao Service
(key([c096d85702a63ee1d350f80977163baf507272ed450ec6fc8a7a7837402bcaa1])).
TrivialGuard("Liberal")) started and waiting for requests
2016/06/18 09:50:25 simpledomainservice: Loaded domain

2016/06/18 09:50:25 simpledomainservice: accepting connections
```

This indicates that the `linux_host`, `simpledomainservice`, `simpleclient` and `simpleserver` have been initialized. The second section shows that the `linux_host` for the soft tao (with the indicated key) has started. The final section indicates that the domain service is started and waiting for request.

Next, you'll notice,

```
TaoParadigm: my name is
key([c096d85702a63ee1d350f80977163baf507272ed450ec6fc8a7a7837402bcaa1])).
.TrivialGuard("Liberal").Program([f4217096352bfe4508d1e2930373df748d35c
ee8f1efa44ac68d0bc973794063])).PolicyKey([b5548720ac9e56c0de44acbcc69c65
e1b6ac07a833eb297e4f846f643bfd7d4c]))
```

1. This is from *simpleserver*, and it is the Tao Principal Name of your *simpleserver* program, running on your Host System, after it has been extended with the hash of the loaded policy key. If you look at the source code for *simpleserver*, you'll notice that the policy key is not embedded in the code; if it had been, the policy key would be reflected in the program measurement. Instead, we read in the policy key key and extend the *simpleserver* Tao Principal Name with its cryptographic hash. The Tao Principal Name is hierarchical. The first segment, "key([c096d85702a63ee1d350f80977...])", describes the host key; it is the sha-256 hash of the host public attestation key in a canonical internal format. Thus, knowing a host's public key (as you typically would, having it's certificate), you could marshal it¹³, hash the marshaled form, and confirm the hash in the Tao name is correct. The second segment, "TrivialGuard("Liberal")" describes the policy the host uses to determine what programs it will run; in this case, the "liberal Guard" will run all programs. In some cases, for example if you "owned" a server, you might have a host which will only run programs signed by you. The third segment, "Program([f421709...])", describes the *simpleserver* program reflecting its measurement. The final segment, "PolicyKey(f3169d...)", describes the policy key using the hash of its public key. Observe that the Tao Principal Name fully reflects all the program code as well as the policy it will execute (as represented by the policy key). In our programs, all signed statements chain up to the policy key. This is the principle policy enforcement mechanism in this style of CloudProxy applications.

For the rest of this description, we will simplify terms like "Program([f421709...])" as "Program(program-measurement)".

Next, notice the statement:

```
simplesdomainservice, speaksfor: key(simpleserver_program_key) speaksfor
key(host-key).Program(simpleserver-measurement).key(policy-key)
```

This is the statement that TaoParadigm will use to request an attestation from the Linux Host System. The resulting Host System supplied attestation is

```
key(hash-of-host-system-key) from notBefore until notAfter says
[simpleserver-program-certificate] speaksfor key(linux-
host).Program(simpleserver-measurement).PolicyKey(hash-of-policy-key)
```

This statement is sent to the domain service which, after checking the measurements and domain policy signs a certificate (with pK_{policy}) that includes the statement

```
key(hash-of-policy-key) from notBefore until notAfter says
[simpleserver-program-certificate] speaksfor keyhash-of-linux-host-
key).Program(simpleserver-measurement).PolicyKey(hash-of-policy-key)
```

¹³ The canonical form is the internal format. See `SerializeEcdsaKeyToInternalName` in `domain_policy/domain_policy.go` or `GetKeyBytes` in `SimpleExampleCpp/taosupport.cc` for examples of how to marshal public keys to hash.

This is the *simpleserver* Program certificate. *Simpleserver*, as we described in the code annotations, stores this certificate, and sealed versions of the corresponding private *simpleserver* ProgramKey and SymmetricKeys. Decrypted and useable versions of these keys are populated in `serverProgramData` by `TaoParadigm`.

After initialization, *simpleserver* waits for client connections.

simpleclient meanwhile, goes through the same *TaoParadigm* initialization (which is not duplicated here) obtaining its Program certificate. *ServerClient* calls `OpenTaoChannel` with its Program Certificate and corresponding key. You'll notice, later in the output, that *simpleserver* opens a secure channel with a peer

```
key(hash-of-linux-host).Program(simple-client-
measurement).PolicyKey(hash-of-policy-key)
```

That peer is just your *simpleclient*. Normally, the Host System on which *simpleclient* runs will be different from the one *simpleserver* runs although in our case, they run on the same host.

Finally, you'll notice that *simpleserver* receives a request

```
2016/02/20 11:27:16      message type: 1
2016/02/20 11:27:16      request_type: SecretRequest
```

and returns the secret which is received by *simpleclient* as

```
simpleclient: secret is
key([c096d85702a63ee1d350f80977163baf507272ed450ec6fc8a7a7837402bcaa1])
.TrivialGuard("Liberal").Program([9711ada89d58b1549a18bc7ed7202b11a6dfc
3c928a70c8b01b4835685975b73]).PolicyKey([b5548720ac9e56c0de44acbcc69c65
e1b6ac07a833eb297e4f846f643bfd7d4c]) 43
```

simpleclient encrypts and integrity protects the secret with its symmetric keys and the process concludes.

We have only discussed the major output elements here. Your output will contain much more including log messages from *simplifiedomainserver*.

The certificate for the *simpleclient* (which is in

`/Domains/domain.simpleexample/SimpleClient/signerCert`) is:

```
Certificate:
  Data:
    Version: 3 (0x2)
    Serial Number: 1455996433984 (0x15300267640)
  Signature Algorithm: ecdsa-with-SHA256
    Issuer: C=US, O=CloudProxy, OU=, ST=WA, CN=SimpleExampleTest
  Validity
    Not Before: Feb 20 19:27:16 2016 GMT
```

```

    Not After : Feb 20 19:27:16 2017 GMT
    Subject: C=US, O=Google,
OU=key([08011001180122450801124104310f3c0d7c5ff1490ace20f167e7de1d5c6847c
84d498c3b4a8087031b49d9a38e7e59f4c5e4f23adc6ce2e394c7ac48923bcfcd7446bba0
f86ef8bbdf89b6d5]).Program([d38d94100ae2bb57cccb97cb347ab060fb28c382bafel
38f253fd19b491b1a15]).key(f3169de17b1032dde230423f7d11dde89c143de147188fa
67acf613d63da0420), ST=, CN=localhost
    Subject Public Key Info:
      Public Key Algorithm: id-ecPublicKey
      Public-Key: (256 bit)
      pub:
        04:7a:d5:40:f6:80:fd:73:a5:80:b8:88:57:7c:60:
        6d:87:b6:78:4a:3f:fc:1c:cc:40:af:34:2d:98:31:
        02:21:02:71:65:66:7f:90:49:91:88:91:21:43:c7:
        f5:50:de:0a:7c:58:c8:6c:10:06:46:fc:3c:1a:a1:
        bb:c7:20:c6:83
      ASN1 OID: prime256v1
    X509v3 extensions:
      X509v3 Key Usage: critical
        Key Agreement, Certificate Sign
      X509v3 Extended Key Usage:
        TLS Web Server Authentication, TLS Web Client
Authentication
  Signature Algorithm: ecdsa-with-SHA256
    30:44:02:20:0c:a0:99:55:79:0d:b7:26:20:07:38:03:da:ba:
    ff:28:0c:fd:94:f6:4e:5f:b1:ad:41:11:89:42:61:fd:5b:e7:
    02:20:7f:26:62:ee:2a:4c:90:e4:f4:7c:d6:c6:2c:b6:1d:db:
    d8:4a:bc:b9:60:26:aa:80:e8:bf:74:bd:ee:34:cb:fe

```

You'll notice that `/Domains/domain.simpleexample/SimpleClient/` also contains the files `sealedsigningKey` (*simpleclient's* sealed program private key), `retrieved_secret` (the "secret" encrypted with *simpleclient's* symmetric keys) and `sealedsymmetricKey` (*simpleclient's* sealed symmetric keys).

Running SimpleExample on real TPM-enabled machines

Complete instructions on installing and deploying Cloudproxy on real TPM enabled machines using KVM, Linux and Docker are in [5]. You should be able to run *simpleexample* on any of those systems.

Upgrade and key management scenarios

Since sealed material is only provided to a Hosted System with exactly the same code identity that sealed the material running on the exact same Host System, while isolated by that Host System, you may be worried about lost data when a Hosted System breaks or becomes unavailable or limitations that may affect key management, software upgrade or distribution when the Hosted System runs on other Host Systems. In fact, it is rather easy to accommodate all these circumstances, and many others, efficiently, securely and in most cases automatically using Cloudproxy, *provided* Cloudproxy applications make provisions for this during development.

Below are a few sever example key management techniques that can be used when a Cloudproxy application is upgraded, a new Cloudproxy application (in the same security domain) is launched, or as applications migrate to other Host Systems. All these mechanisms preserve the confidentiality and integrity of all Cloudproxy applications and their data.

There is a discussion of many of the mechanisms, as they might affect client software used across different security domains, by users with no control over the application code while supporting consumer transparency (the most challenging case) in [4]. Here we restrict ourselves to cooperating server applications for simplicity.

To ease description, imagine all application data is stored locally or remotely and probably redundantly in encrypted, integrity protected files. Each file is encrypted and integrity protected with individual file keys and each file key is itself encrypted and integrity protected with a group sealing keys. Different groups of file keys are protected by different sealing keys to reduce the risk of universal compromise. Every key has exposed meta data consisting of a globally unique name for the entity it protects, the key type and an “epoch.” Epochs increases monotonically as the keys are rotated¹⁴. As keys for a new epoch become available, the objects they protect are re-encrypted, over a reasonable period of time (the Rotation Period). During this time, keys for the prior epoch are available and can be used to decrypt objects; however, as soon as new epoch keys are available, all new data is encrypted with the new epoch keys. At the end of the Rotation Period, once applications have confirmed that all data is protected with the keys from the most recent epoch, old epoch keys are deprecated.

The first option to deal with “brittle keys” protecting application data is standard: use a distributed key server which authenticates a Tao Program and provisions symmetric keys for data over the authenticated, encrypted, integrity protected Tao Channel. In this case, Cloudproxy applications do not locally store data protection keys¹⁵ but contact a key server (over a Tao Channel). The key server (which does key rotation, etc., as many do) authenticates the Hosted System that needs keys and verifies that it is authorized to receive those keys, and if so, they are transmitted over the Tao Channel. Hosted Systems can be upgraded and all authorization policy can be maintained by the key service in this model. Note that application upgrade is automatic when you use this option even when the policy keys change: New versions of Hosted Systems simply re-initialize (get new program keys and certificates) using the (centralized or distributed) security domain service and no special provision, aside from current policy at the security domain service, need be provided¹⁶.

Sample code for such a keyserver is

`$CLOUDPROXY/go/support_infrastructure/CPSecretService`. Support libraries that allow you to build symmetric key trees (and partitioned secrets) with key rotation support in the style

¹⁴ And you certainly should rotate keys as part of effective cryptographic hygiene!

¹⁵ Although they may cache such keys --- see below.

¹⁶ Many events may cause such a policy change including a determination that previously trusted hardware elements have been compromised.

suggested above are in `$CLOUDPROXY/go/support_libraries/protected_objects, rotation_support`. This mechanism allows you to cache sealed keys using the key tree to avoid contacting a keyserver on each activation but you will likely want to add a “key change” notification as new key epochs are introduced.

An alternative, less centralized, key rotation mechanism, which we call certificate based key disclosure, allows individual Hosted Systems maintaining their own keys to protect files as well as perform key rotation themselves. When software is upgraded or new programs are introduced, the new programs or upgraded programs come with a certificate signed by the policy key that instruct one Hosted System to disclose these keys to the new version (or new) Hosted System. Since this can result in lost data if a Host System becomes unavailable, Hosted Systems would likely distribute these keys to different instances on different machines to ensure continuity.

Support libraries that implement certificate based key disclosure are in `$CLOUDPROXY/go/support_libraries/secret_disclosure_support`.

Finally, when new data protection keys are established for an application task, Hosted Systems can contact a domain service to receive intermediate keys for registered files or file classes. These keys can be sealed using the Host System provided Seal and used without contacting the service each time the Hosted System starts. This mechanism places additional administrative burden on each Hosted System to contact the “key sharing service” as intermediate keys rotate but this is not uncommon.

It is important to note that while the foregoing descriptions treat keys as “all or nothing” entities, all these scheme have corresponding “split key” implementations to achieve higher security. In addition, any security domain may elect to have an authorized Cloudproxy Hosted System archive data. Such an archive application, upon which security domain policy confers access to data, can, in the background, archive data to (centralized or distributed) repositories. There are many other possible mechanisms to do key management but these should get you started.

There is a further discussion of key management in a Cloudproxy environment in [5] as it affects deployment of Cloudproxy applications.

Suggested Exercises

That’s all there is to using Cloudproxy. Here are some suggested exercises to complete the training:

1. Write a more complicated set of domain applications; for example, see `“$CLOUDPROXYDIR/go/apps/fileproxy.”`
2. Boot a Linux Host System on TPM supported hardware using the TPM to root the Linux Tao (see ... for instructions).

3. Boot a KVM Host System on tpm supported hardware and then run a stacked VM host in a Linux partition (see ... for instructions). SimpleExample should run fine in the VM(s) with slight changes to the initialization scripts.
4. Explore the Data log engine (examples?)
5. What happens if you make a modification to simpleclient.go and immediately run it on `linux_host` without reinitializing the domain?
6. Write and compile some Datalog rules to do some fancier authorization and try it.
7. Understand how to start Hosted Systems by studying
“\$CLOUDPROXYDIR/go/apps/tao_launch.”
8. Correct any errors in this paper or the examples and send the corrections or suggestions to us.
9. Write an awesome Cloudproxy based application and tell us and you friends all about it.
10. Repeat step 9 and have fun!

References

- [1] Manferdelli, Roeder, Schneider, The CloudProxy Tao for Trusted Computing, <http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-135.pdf>.
- [2] CloudProxy Source code, <http://github.com/jlmucb/cloudproxy>. Kevin Walsh and Tom Roeder were principal authors of the Go version.
- [3] TCG, TPM specs, http://www.trustedcomputinggroup.org/resources/tpm_library_specification
- [4] Beekman, Manferdelli, Wagner, Attestation Transparency: Building secure Internet services for legacy clients. AsiaCCS, 2016.
- [5] Manferdelli, Roeder, Telang, Nuts and Bolts of Deploying Real Cloudproxy Applications. Doc directory in [2].

CloudProxy's Authorization Language

Package *auth* implements Tao authorization and authentication, by defining and implementing a logic for describing principals, their trust relationships, and their beliefs.

The grammar for a formula in the logic is roughly:

```
Form ::= Term [from Time] [until Time] says Form
      | Term speaksfor Term
      | forall TermVar : Form
      | exists TermVar : Form
      | Form implies Form
      | Form or Form or ...
      | Form and Form and ...
      | not Form
      | Pred | false | true
```

Quantification variables range over Terms.

```
TermVar : Identifier
```

Times are integers interpreted as 64-bit Unix timestamps.

```
Time ::= int64
```

Predicates are like boolean-valued pure functions, with a name and zero or more terms as arguments.

```
Pred ::= Identifier(Term, Term, ...)
      | Identifier()
```

Terms are concrete values, like strings, integers, or names of principals.

```
Term ::= Str | Bytes | Int | Prin | PrinTail | TermVar
```

Int can be any Go int. Str is a double-quoted Go string. Bytes is written as pairs of hex digits, optionally separated by whitespace, between square brackets. Bytes can also be written as base64w without whitespace between curly braces.

Principal names specify a key or a tpm, and zero or more extensions to specify a sub-principal of that key.

```
PrinType ::= key | tpm
Prin ::= PrinType(Term)
      | PrinType(Term).PrinExt.PrinExt...
PrinExt ::= Identifier(Term, Term, ...)
          | Identifier()
```

Principal tails represent a sequence of extensions that are not rooted in a principal. They are used to make statements about authorized extensions independent of the root principal. For example, they are used to specify that a given program is authorized to execute on any platform. A PrinTail must be followed by at least one extension.

```
PrinTail ::= ext.PrinExt.PrinExt...
```

Identifiers for predicate and principal extension names and quantification variables are limited to simple ascii printable identifiers, with initial upper-case, and no punctuation except '_':

```
PredName ::= [A-Z][a-zA-Z0-9_]*  
ExtName  ::= [A-Z][a-zA-Z0-9_]*
```

The keywords used in the above grammar are:

```
from, until, says, speakfor, forall, exists, implies, or, and, not, false,  
true, key
```

The punctuation used are those for strings and byte slices, plus:

```
'(', ')', ',', '.', ':'
```

It is possible to represent nonsensical formulas, so some sanity checking maybe called for. For example, in general:

1. The left operand of Says should be Prin or TermVar, as should both operands of Speaksfor.
2. All TermVar variables should be bound.
3. Conjunctions should have at least one conjunct.
4. Disjunctions should have at least one disjunct.
5. Identifiers should be legal using the above rules.
6. The parameter for key() should be TermVar or Bytes.

Specific applications may impose additional restrictions on the structure of formulas they accept.

All of the above elements have three distinct representations. The first representation is ast-like, with each element represented by an appropriate Go type, e.g. an int, a string, or a struct containing pointers (or interfaces) for child elements. This representation is meant to be easy to programmatically construct, split apart using type switches, rearrange, traverse, etc.

The second representation is textual, which is convenient for humans but isn't canonical and can involve tricky parsing. When parsing elements from text:

Whitespace is ignored between elements (except around the subprincipal dot operator, and before the open paren of a Pred, Prin, or, PrinExt) operator, and before the open parenthesis of a Pred, Prin, or, PrinExt).

For binary operators taking two Forms, the above list shows the productions in order of increasing precedence. In all other cases, operations are parsed left to right. Parenthesis can be used for specifying precedence explicitly.

When pretty-printing elements to text, a single space is used before and after keywords, commas, and colons. Elements can also be pretty-printed with elision, in which case keys and long strings are truncated.

The third representation is an encoded sequence of bytes. This is meant to be compact,

relatively easy to parse, and suitable for passing over sockets, network connections, etc. The encoding format is custom-designed, but is roughly similar to the format used by protobuf.

The encoding we use instead is meant to be conceptually simple, reasonably space efficient, and simple to decode. And unlike most of the other schemes above, strictness rather than flexibility is preferred. For example, when decoding a Form used for authorization, unrecognized fields should not be silently skipped, and unexpected types should not be silently coerced.

Each element is encoded as a type tag followed by encodings for one or more values. The tag is encoded as a plain (i.e. not zig-zag encoded) varint, and it determines the meaning, number, and types of the values. Values are encoded according to their type:

An integer or bool is encoded as plain varint.

A string is encoded as a length (plain varint) followed by raw bytes.

A pointer is encoded the same as a boolean optionally followed by a value.

Variable-length slices (e.g. for conjuncts, disjuncts, predicate arguments) are encoded as a count (plain varint) followed by the encoding for each element.

An embedded struct or interface is encoded as a tag and encoded value.

Differences from protobuf: Our tags carry implicit type information. In protobuf, the low 3 bits of each tag carries an explicit type marker. That allows protobuf to skip over unrecognized fields (not a design goal for us). It also means protobuf can only handle 15 unique tags before overflowing to 2 byte encodings.

Our tags describe both the meaning and the type of all enclosed values, and we use tags only when the meaning or type can vary (i.e. for interface types). Protobuf uses tags for every enclosed value, and those tags also carry type information. Protobuf is more efficient when there are many optional fields. For us, nearly all fields are required.

Enclosed values in our encoding must appear in order. Protobuf values can appear in any order. Protobuf encodings can concatenated, truncated, etc., all non-features for us.

Note: In most cases, a tag appears only when the type would be ambiguous, i.e. when encodingTerm or Form.

Further information on auth functions appears in a later appendix.

SimpleExample in C++

You can also write your CloudProxy protected programs in C++. This section describes how to do it using the code example in SimpleClientCpp which implements the very same functionality the go simpleclient does but in C++

Installing C++ libraries (This needs to be fixed)

Before compiling and linking your C++ program, you must compile the standard Cloudproxy C++ libraries required. To do this:

```
cd $CLOUDPROXYDIR/src
cmake .
make
make install
```

This will produce `taoauth.a` which implements the authorization support routines we will use.

The C++ interfaces use program generated C++ classes constructed by a go program which are in turn compiled into a library. This reduces the chances that the Go support will diverge from the C++ support. To build this type:

```
go run ${CMAKE_SOURCE_DIR}/../go/apps/genauth/genauth/genauth.go
-ast_file ${CMAKE_SOURCE_DIR}/../go/tao/auth/ast.go -binary_file
${CMAKE_SOURCE_DIR}/../go/tao/auth/binary.go -header_file
${CMAKE_BINARY_DIR}/auth.h -impl_file ${CMAKE_BINARY_DIR}/auth.cc
```

This produces the files `auth.cc` and `auth.h` which are compiled into the `libauth.a` library. You also need to compile some string support from chromium. This code (and associated build files) are in `$CLOUDPROXY/src/third_party/chromium`. This produces the library `libchromium.a`.

You will need to link `taoauth.a`, `libauth.a` and `libchromium.a` in your C++ Cloudproxy program and, you will have to include “.pb.h” files and compile corresponding “.pb.cc” files produced by protoc. You’ll see examples of all this in the SimpleClientCpp example code and makefiles.

SimpleClient in C++

We only implement *simpleclient* in C++. The reader can make the obvious changes to implement a corresponding *simpleserver* or *simpleservice*.

The C++ *simpleclient* source code is in

`$CLOUDPROXY/go/apps/simpleexample/SimpleClientCpp` and is structured in the same way as the Go version. Most of the Tao related support code is in `taosupport.cc` and an associated include file `taosupport.h`; this corresponds to `taosupport.go` in the Go version. The main loop is in `simpleclient_cc.cc`. The main loop is nearly self-explanatory: it simply calls support functions in `taosupport` to create a Tao program data object (as in go), initialize the program data object (using `InitTao` in `taosupport.cc`), open the TaoChannel (using `OpenTaoChannel` in `taosupport.cc`), sends a request for its secret (using `SendRequest` in `taosupport.cc`), receiving the response (using `SendRequest` in `taosupport.cc`) and finally printing out the retrieved secret. So we'll confine our comments to the code in `taosupport.cc`. This should help explain how to use CloudProxy in C++ programs.

Running simpleclient_cc

First compile `simpleclient_cc.exe` by typing

```
make -f simpleclient.mak
```

This makefile puts the executable in `/Domains`.

We will use the same *simpleserver* and *simplifiedomainservice* we used for the Go version of *simpleexample* and simply substitute *simpleclient_cc.exe* for *simpleclient* when we run the programs. As with the go version, we provide scripts to run the programs:

```
#do the following ONLY if you haven't run the Go
# version in this session.
cd $CLOUDPROXY/go/apps/simpleexample/SimpleDomain
./gentemplate
./initdomainstorage
./initkey
sudo bash
# First make sure previous hosts were killed and the admin_socket
# was freed.
# Now run the programs
./runallcc
```

The resulting output should be similar to the Go output and for the same reasons! Here is an example:

```
DomainRequest
2016/06/18 09:50:49 DomainRequest
IsAuthenticationValid
2016/06/18 09:50:49 simplifiedomainservice, IsAuthenticationValid name is
key([c096d85702a63ee1d350f80977163baf507272ed450ec6fc8a7a7837402bcaal]).T
rivialGuard("Liberal").Program([e458cd8a843494d1388f44b7a0da5bbd17b14abae
73b0b61ed06f1428c989399]).PolicyKey([b5548720ac9e56c0de44acbcc69c65e1b6ac
07a833eb297e4f846f643bfd7d4c])
2016/06/18 09:50:49 simplifiedomainservice, IsAuthenticationValid returning
true
```

```

SimpleDomainService: key principal:
key([a073a070f2263eb17dc60c0b3c1b9769e141222e3a84bd35392b69e6268ac3d6]),
program principal:
key([c096d85702a63ee1d350f80977163baf507272ed450ec6fc8a7a7837402bcaal]).T
rivialGuard("Liberal").Program([e458cd8a843494d1388f44b7a0da5bbd17b14abae
73b0b61ed06f1428c989399]).PolicyKey([b5548720ac9e56c0de44acbcc69c65elb6ac
07a833eb297e4f846f643bfd7d4c])

simpleclientcpp: Simple client name:
key(c096d85702a63ee1d350f80977163baf507272ed450ec6fc8a7a7837402bcaal).Tri
vialGuard().Program(e458cd8a843494d1388f44b7a0da5bbd17b14abae73b0b61ed06f
1428c989399).PolicyKey(b5548720ac9e56c0de44acbcc69c65elb6ac07a833eb297e4f
846f643bfd7d4c)

2016/06/18 09:50:49 server, peer client name:
key([c096d85702a63ee1d350f80977163baf507272ed450ec6fc8a7a7837402bcaal]).T
rivialGuard("Liberal").Program([e458cd8a843494d1388f44b7a0da5bbd17b14abae
73b0b61ed06f1428c989399]).PolicyKey([b5548720ac9e56c0de44acbcc69c65elb6ac
07a833eb297e4f846f643bfd7d4c])
2016/06/18 09:50:49 server: at accept
simpleclient: established Tao Channel with
Sent request
2016/06/18 09:50:49 serviceThread, got message:
2016/06/18 09:50:49 Message
2016/06/18 09:50:49     message type: 1
2016/06/18 09:50:49     request_type: SecretRequest
2016/06/18 09:50:49     data:
2016/06/18 09:50:49
2016/06/18 09:50:49 HandleServiceRequest response buffer:
2016/06/18 09:50:49 Message
2016/06/18 09:50:49     message type: 2

2016/06/18 09:50:49     request_type: SecretRequest
simpleclient: secret is
key([c096d85702a63ee1d350f80977163baf507272ed450ec6fc8a7a7837402bcaal]).T
rivialGuard("Liberal").Program([e458cd8a843494d1388f44b7a0da5bbd17b14abae
73b0b61ed06f1428c989399]).PolicyKey([b5548720ac9e56c0de44acbcc69c65elb6ac
07a833eb297e4f846f643bfd7d4c])43, done
simpleserver: client thread terminating

```

Simple Domain Protocol

The Domain Protocol is used in the transmission of a hosted system's attestation which includes the TaoName of the requesting client (like

```
key([c096d85702a63ee1d350f80977163baf507272ed450ec6fc8a7a7837402bcaa1]).TrivialGuard("Liberal").Program([e458cd8a843494d1388f44b7a0da5bbd17b14abae73b0b61ed06f1428c989399]).PolicyKey([b5548720ac9e56c0de44acbcc69c65e1b6ac07a833eb297e4f846f643bfd7d4c])). Here is a brief description:
```

1. A client (*SimpleServer*, *SimpleClient*, *simpleclient_cc.exe*) sends the Domain Service a `DomainCertRequest` which includes a key type, a DER encoded PKIX public key description of its program key along with delegation signed by the host that says:
key(hash-of-host-key-in-internal-key-format) says key(hash-of-program-key-in-internal-key-format) speaks for current-tao-program-name.
2. *SimpleDomainService* gets the public-key from the DER encoded key description and converts it into the internal serialized key form which it hashes it using SHA-256. It then confirms the hash of the program key is the same as in the corresponding delegation. It checks the host key is trusted and the program-name corresponds to an approved program. Using the key information extracted from the delegation and the DER encoded key description it signs the program certificate with the policy-key. This certificate as above, has the Tao Program Name in the cert organizational unit. We do not need a delegation but if we did, it would be:
key(hash-of-policy-key-in-internal-key-format) says key(hash-of-program-key-in-internal-key-format) speaksfor Tao-Program-name
3. It returns the signed certificate in a `DomainCertResponse`.

All these names are well specified and can be calculated (and hence confirmed) using just a standard hash function. If someone wanted to calculate these offline with another program, the "specification" is simple, new key types can be easily accommodated and Certificate validation is standard. Key names and delegations are always compact, the name lengths don't depend on the key types at all, even for a quantum public key system.

Core Cloudproxy Functions

This and the next two appendices describe the native functions available in the Go version of Cloudproxy. In the code sample, we duplicated some of them for simplicity. The first sections contains functions you may call directly in more sophisticated Cloudproxy applications. Some will be familiar from the sample code.

`./attestation.go`: These functions handle attestations.

- `func (a *Attestation) ValidSigner() (auth.Prin, error)`
- `func (a *Attestation) Validate() (auth.Says, error)`
- `func GenerateAttestation(s *Signer, delegation []byte, stmt auth.Says) (*Attestation, error)`

`./ca.go`: These functions are employed by or when communicating with a domain service.

- `func HandleCARRequest(conn net.Conn, s *Signer, guard Guard)`
- `func RequestAttestation(network, addr string, keys *Keys, v *Verifier) (*Attestation, error)`
- `func RequestTruncatedAttestation(network, addr string, keys *Keys, v *Verifier) (*Attestation, error)`
- `func RequestDatalogRules(network, addr string, v *Verifier) (*DatalogRules, error)`
- `func RequestACLSet(network, addr string, v *Verifier) (*ACLSet, error)`

`./client.go`: These are networking support functions called by a “client.”

- `func EncodeTLSCert(keys *Keys) (*tls.Certificate, error)`
- `func ListenTLS(network, addr string) (net.Listener, error)`
- `func DialTLS(network, addr string) (net.Conn, error)`
- `func DialTLSWithKeys(network, addr string, keys *Keys) (net.Conn, error)`
- `func DialWithNewX509(network, addr string, guard Guard, v *Verifier) (net.Conn, error)`
- `func Dial(network, addr string, guard Guard, v *Verifier, keys *Keys) (net.Conn, error)`
- `func AddEndorsements(guard Guard, a *Attestation, v *Verifier) error`
- `func TruncateAttestation(kprin auth.Prin, a *Attestation) (auth.Says, auth.PrinExt, error)`
- `func IdenticalDelegations(s, t auth.Form) bool`

`./listener.go`: These are networking support functions called by a “server.”

- `func Listen(network, laddr string, config *tls.Config, g Guard, v *Verifier, del *Attestation) (net.Listener, error)`
- `func ListenAnonymous(network, laddr string, config *tls.Config, g Guard, v *Verifier, del *Attestation) (net.Listener, error)`
- `func ValidatePeerAttestation(a *Attestation, cert *x509.Certificate, guard Guard) error`
- `func (l *listener) Accept() (net.Conn, error)`
- `func (l *anonymousListener) Accept() (net.Conn, error)`
- `func (l *listener) Close() error`
- `func (l *listener) Addr() net.Addr`

`./root_host.go`: These are host interface functions. Each different host type duplicates some of these.

- `func NewTaoRootHostFromKeys(k *Keys) (Host, error)`
- `func NewTaoRootHost() (Host, error)`
- `func (t *RootHost) GetRandomBytes(childSubprin auth.SubPrin, n int) (bytes []byte, err error)`
- `func (t *RootHost) GetSharedSecret(tag string, n int) (bytes []byte, err error)`
- `func (t *RootHost) Attest(childSubprin auth.SubPrin, issuer *auth.Prin,`
- `func (t *RootHost) Encrypt(data []byte) (encrypted []byte, err error)`
- `func (t *RootHost) Decrypt(encrypted []byte) (data []byte, err error)`
- `func (t *RootHost) AddedHostedProgram(childSubprin auth.SubPrin) error`
- `func (t *RootHost) RemovedHostedProgram(childSubprin auth.SubPrin) error`

- func (t *RootHost) HostName() auth.Prin

./tao.go:

- func Register(name string, generator func(string) (Tao, error))
- func ParentFromConfig(tc Config) Tao
- func Parent() Tao

./domain.go: These functions are for domain handling. Note that we did our own domain handling in the C++ implementation.

- func (cfg *DomainConfig) SetDefaults()
- func (d *Domain) String() string
- func (d *Domain) Subprincipal() auth.SubPrin
- func CreateDomain(cfg DomainConfig, configPath string, password []byte) (*Domain, error)
- func (d *Domain) Save() error
- func LoadDomain(configPath string, password []byte) (*Domain, error)
- func (d *Domain) ExtendTaoName(tao Tao) error
- func (d *Domain) RulesPath() string

./keys.go: These functions handle key interface (Todo: Currently, only ECC based asymmetric algorithms are supported).

- func (v *Verifier) ToPrincipal() auth.Prin
- func FromPrincipal(prin auth.Prin) (*Verifier, error)
- func (s *Signer) Sign(data []byte, context string) ([]byte, error)
- func (s *Signer) GetVerifier() *Verifier
- func (v *Verifier) Verify(data []byte, context string, sig []byte) (bool, error)
- func FromX509(cert *x509.Certificate) (*Verifier, error)
- func (v *Verifier) Equals(cert *x509.Certificate) bool
- func (c *Crypter) Encrypt(data []byte) ([]byte, error)
- func (c *Crypter) Decrypt(ciphertext []byte) ([]byte, error)
- func (s *Signer) ToPrincipal() auth.Prin
- func (s *Signer) CreateSelfSignedDER(name *pkix.Name) ([]byte, error)
- func (s *Signer) CreateSelfSignedX509(name *pkix.Name) (*x509.Certificate, error)

You will seldom call the following functions since they are called primarily as helpers or by utility functions.

./soft_tao.go:

- func NewSoftTao(path string, password []byte) (Tao, error)
- + root host

./stacked_host.go:

- func NewTaoStackedHostFromKeys(k *Keys, t Tao) (Host, error)
- func NewTaoStackedHost(t Tao) (Host, error)
- plus same as root host

./keys.go

- func GenerateSigner() (*Signer, error)
- func MarshalSignerDER(s *Signer) ([]byte, error)
- func UnmarshalSignerDER(signer []byte) (*Signer, error)
- func NewX509Name(p *X509Details) *pkix.Name

-
- func (s *Signer) CreateCRL(cert *x509.Certificate, revokedCerts []pkix.RevokedCertificate, now, expiry time.Time) ([]byte, error)
- func (s *Signer) CreateSignedX509(caCert *x509.Certificate, certSerial int, subjectKey *Verifier, subjectName *pkix.Name) (*x509.Certificate, error)
- func MarshalSignerProto(s *Signer) (*CryptoKey, error)
- func MarshalPublicSignerProto(s *Signer) *CryptoKey
- func MarshalVerifierProto(v *Verifier) *CryptoKey
- func UnmarshalSignerProto(ck *CryptoKey) (*Signer, error)
- func (s *Signer) CreateHeader() (*CryptoHeader, error)
- func UnmarshalVerifierProto(ck *CryptoKey) (*Verifier, error)
- func (v *Verifier) CreateHeader() (*CryptoHeader, error)
- func GenerateCrypter() (*Crypter, error)
- func UnmarshalCrypterProto(ck *CryptoKey) (*Crypter, error)
- func (c *Crypter) CreateHeader() (*CryptoHeader, error)
- func GenerateDeriver() (*Deriver, error)
- func (d *Deriver) Derive(salt, context, material []byte) error
- func MarshalDeriverProto(d *Deriver) (*CryptoKey, error)
- func UnmarshalDeriverProto(ck *CryptoKey) (*Deriver, error)
- func (k *Keys) X509Path() string
- func (k *Keys) PBEKeysetPath() string
- func (k *Keys) PBESignerPath() string
- func (k *Keys) SealedKeysetPath() string
- func (k *Keys) PlaintextKeysetPath() string
- func ZeroBytes(b []byte)
- func NewTemporaryKeys(keyTypes KeyType) (*Keys, error)
- func NewSignedOnDiskPBEKeys(keyTypes KeyType, password []byte, path string, name *pkix.Name, serial int, signer *Keys) (*Keys, error)
- func NewOnDiskPBEKeys(keyTypes KeyType, password []byte, path string, name *pkix.Name) (*Keys, error)
- func NewTemporaryTaoDelegatedKeys(keyTypes KeyType, t Tao) (*Keys, error)
- func PBEEncrypt(plaintext, password []byte) ([]byte, error)
- func PBEDecrypt(ciphertext, password []byte) ([]byte, error)
- func MarshalKeyset(k *Keys) (*CryptoKeyset, error)
- func UnmarshalKeyset(cks *CryptoKeyset) (*Keys, error)
- func NewOnDiskTaoSealedKeys(keyTypes KeyType, t Tao, path, policy string) (*Keys, error)
- func (k *Keys) Save(t Tao) error
- func LoadKeys(keyTypes KeyType, t Tao, path, policy string) (*Keys, error)
- func (k *Keys) NewSecret(file string, length int) ([]byte, error)
- func SaveKeyset(k *Keys, dir string) error

./linux_host_admin_rpc.go:

- func NewLinuxHostAdminClient(conn *net.UnixConn) LinuxHostAdminClient
- func (client LinuxHostAdminClient) StartHostedProgram(spec *HostedProgramSpec) (auth.SubPrin, int, error)
- func (client LinuxHostAdminClient) StopHostedProgram(subprin auth.SubPrin) error
- func (client LinuxHostAdminClient) ListHostedPrograms() (name []auth.SubPrin, pid []int, err error)
- func (client LinuxHostAdminClient) WaitHostedProgram(pid int, subprin auth.SubPrin) (int, error)
- func (client LinuxHostAdminClient) KillHostedProgram(subprin auth.SubPrin) error
- func (client LinuxHostAdminClient) HostName() (auth.Prin, error)
- func (client LinuxHostAdminClient) Shutdown() error
- func NewLinuxHostAdminServer(host *LinuxHost) LinuxHostAdminServer
- func (server LinuxHostAdminServer) Serve(sock *net.UnixListener) error

- func (server linuxHostAdminServerStub) StartHostedProgram(r *LinuxHostAdminRPCRequest, s *LinuxHostAdminRPCResponse) error
- func (server linuxHostAdminServerStub) StopHostedProgram(r *LinuxHostAdminRPCRequest, s *LinuxHostAdminRPCResponse) error
- func (server linuxHostAdminServerStub) ListHostedPrograms(r *LinuxHostAdminRPCRequest, s *LinuxHostAdminRPCResponse) error
- func (server linuxHostAdminServerStub) WaitHostedProgram(r *LinuxHostAdminRPCRequest, s *LinuxHostAdminRPCResponse) error
- func (server linuxHostAdminServerStub) KillHostedProgram(r *LinuxHostAdminRPCRequest, s *LinuxHostAdminRPCResponse) error
- func (server linuxHostAdminServerStub) HostName(r *LinuxHostAdminRPCRequest, s *LinuxHostAdminRPCResponse) error
- func (server linuxHostAdminServerStub) Shutdown(r *LinuxHostAdminRPCRequest, s *LinuxHostAdminRPCResponse) error

./linux_host_tao_rpc.go:

- func NewLinuxHostTaoServer(host *LinuxHost, child *LinuxHostChild) LinuxHostTaoServer
- func (server LinuxHostTaoServer) Serve(conn io.ReadWriteCloser) error

./rpc.go:

- func DeserializeRPC(s string) (*RPC, error)
- func DeserializeFileRPC(s string) (*RPC, error)
- func DeserializeUnixSocketRPC(p string) (*RPC, error)
- func NewRPC(rwc io.ReadWriteCloser, serviceName string) (*RPC, error)
- func (t *RPC) GetTaoName() (auth.Prin, error)
- func (t *RPC) ExtendTaoName(subprin auth.SubPrin) error
- func (t *taoRandReader) Read(p []byte) (n int, err error)
- func (t *RPC) Rand() io.Reader
- func (t *RPC) GetRandomBytes(n int) ([]byte, error)
- func (t *RPC) GetSharedSecret(n int, policy string) ([]byte, error)
- func (t *RPC) Attest(issuer *auth.Prin, time, expiration *int64, message auth.Form) (*Attestation, error)
- func (t *RPC) Seal(data []byte, policy string) (sealed []byte, err error)
- func (t *RPC) Unseal(sealed []byte) (data []byte, policy string, err error)

Auth Functions

./auth/ast.go:

- func (f Says) Commences() bool
- func (f Says) Expires() bool
- func (t Int) Identical(other Term) bool
- func (t Str) Identical(other Term) bool
- func (t Bytes) Identical(other Term) bool
- func (t Prin) Identical(other Term) bool
- func (t PrinTail) Identical(other Term) bool
- func (t TermVar) Identical(other Term) bool
- func (e PrinExt) Identical(other PrinExt) bool
- func (t SubPrin) Identical(other SubPrin) bool
- func SubprinOrIdentical(child, parent Term) bool
- func (t Prin) MakeSubprincipal(e SubPrin) Prin
- func MakePredicate(name string, arg ...interface{}) Pred
- func NewKeyPrin(material []byte) Prin
- func Marshal(e LogicElement) []byte
- func (t Prin) Marshal(buf *Buffer)
- func (t PrinTail) Marshal(buf *Buffer)
- func (t PrinExt) Marshal(buf *Buffer)
- func (s SubPrin) Marshal(buf *Buffer)
- func (t Str) Marshal(buf *Buffer)
- func (t Bytes) Marshal(buf *Buffer)
- func (t Int) Marshal(buf *Buffer)
- func (t TermVar) Marshal(buf *Buffer)
- func (f Pred) Marshal(buf *Buffer)
- func (f Const) Marshal(buf *Buffer)
- func (f Not) Marshal(buf *Buffer)
- func (f And) Marshal(buf *Buffer)
- func (f Or) Marshal(buf *Buffer)
- func (f Implies) Marshal(buf *Buffer)
- func (f Speaksfor) Marshal(buf *Buffer)
- func (f Says) Marshal(buf *Buffer)
- func (f Forall) Marshal(buf *Buffer)
- func (f Exists) Marshal(buf *Buffer)
- func UnmarshalPrin(bytes []byte) (p Prin, err error)
- func UnmarshalPrinTail(bytes []byte) (p PrinTail, err error)
- func UnmarshalPrinExt(bytes []byte) (p PrinExt, err error)
- func UnmarshalTerm(bytes []byte) (Term, error)
- func UnmarshalSubPrin(bytes []byte) (SubPrin, error)
- func UnmarshalForm(bytes []byte) (Form, error)
- func unmarshalForm(buf *Buffer) (Form, error)
- func (buf *Buffer) Bytes() []byte
- func (buf *Buffer) EncodeVarint(i int64)
- func (buf *Buffer) DecodeVarint() (int64, error)
- func (buf *Buffer) EncodeBool(b bool)
- func (buf *Buffer) DecodeBool() (bool, error)
- func (buf *Buffer) EncodeString(s string)
- func (buf *Buffer) DecodeString() (string, error)
- func (p Prin) Format(out fmt.State, verb rune)

- func (p PrinTail) Format(out fmt.State, verb rune)
- func (e PrinExt) Format(out fmt.State, verb rune)
- func (p SubPrin) Format(out fmt.State, verb rune)
- func (t Str) Format(out fmt.State, verb rune)
- func (t Bytes) Format(out fmt.State, verb rune)
- func (t Int) Format(out fmt.State, verb rune)
- func (t TermVar) Format(out fmt.State, verb rune)
- func (f Pred) Format(out fmt.State, verb rune)
- func (f Const) Format(out fmt.State, verb rune)
- func (f Not) Format(out fmt.State, verb rune)
- func (f And) Format(out fmt.State, verb rune)
- func (f Or) Format(out fmt.State, verb rune)
- func (f Implies) Format(out fmt.State, verb rune)
- func (f Speaksfor) Format(out fmt.State, verb rune)
- func (f Says) Format(out fmt.State, verb rune)
- func (f Forall) Format(out fmt.State, verb rune)
- func (f Exists) Format(out fmt.State, verb rune)

Guard functions

These are the guard interface functions. We show them for the Datalog guard only.

`./datalog_guard.go:`

- `func (sp *subprinPrim) String() string`
- `func (sp *subprinPrim) Assert(c *datalog.Clause) error`
- `func (sp *subprinPrim) Retract(c *datalog.Clause) error`
- `func (sp *subprinPrim) Search(target *datalog.Literal, discovered func(c *datalog.Clause))`
- `func NewTemporaryDatalogGuard() Guard`
- `func NewDatalogGuardFromConfig(verifier *Verifier, config DatalogGuardDetails) (*DatalogGuard, error)`
- `func NewDatalogGuard(verifier *Verifier) *DatalogGuard`
- `func (g *DatalogGuard) Subprincipal() auth.SubPrin`
- `func (g *DatalogGuard) ReloadIfModified() error`
- `func (g *DatalogGuard) GetSignedDatalogRules(signer *Signer) (*SignedDatalogRules, error)`
- `func (g *DatalogGuard) Save(signer *Signer) error`
- `func (g *DatalogGuard) Authorize(p auth.Prin, op string, args []string) error`
- `func (g *DatalogGuard) Retract(p auth.Prin, op string, args []string) error`
- `func (g *DatalogGuard) IsAuthorized(p auth.Prin, op string, args []string) bool`
- `func (g *DatalogGuard) AddRule(rule string) error`
- `func (g *DatalogGuard) RetractRule(rule string) error`
- `func (g *DatalogGuard) Clear() error`
- `func (g *DatalogGuard) Query(query string) (bool, error)`
- `func (g *DatalogGuard) RuleCount() int`
- `func (g *DatalogGuard) GetRule(i int) string`
- `func (g *DatalogGuard) RuleDebugString(i int) string`
- `func (g *DatalogGuard) String() string`